/\* -----------------------------------------------

Name: J-Zach Loke

Course: CMPS-385

Semester: Spring 2020

Project: No. 7 Part 1

Purpose: Write a program to display the union of two given sets and

insert the elements of each set into an ordered linked list

before finding their union.

----------------------------------------------- \*/

#include <iostream>

class OrderedList

{

private:

struct node

{

int val;

node \*next;

};

node \*list;

public:

OrderedList() { list = NULL; }

void push(int x)

{

/\* name: push

input: int x

output: N/A

purpose: adds an element into the linked list stack and makes sure to keep it ordered (smallest to largest). \*/

// create the node element we wish to insert

node \*insert = new(node);

insert->val = x;

// temporary variables to help traversal

node \*prev = list; // represents the node we've previously examined (needed because we can't traverse backwards).

node \*curr = list; // represents the current node we're examining.

while (true)

{

// determines where in the list to insert the new node.

// ordered from smallest to largest.

if (curr == NULL)

{

// executes if we've reached the end of the list.

// if this executes, that means all nodes in the list have a smaller value than the node we're inserting.

// in this case, we should append the new node to the end of the list.

// also executes if the list is empty (ie. the first element is NULL)

if (list == NULL) { list = insert; }

else { prev->next = insert; }

insert->next = NULL;

return;

}

else if (curr->val > insert->val)

{

// executes if the node we're examining has a value larger than the node we're inserting.

// if this executes, that means we've found the correct place to insert the new node.

if (prev == curr) { list = insert; }

else { prev->next = insert; }

insert->next = curr;

return;

}

// executes when we haven't found the correct place to insert the new node.

prev = curr;

curr = curr->next;

}

}

int pop()

{

/\* name: pop

input: N/A

output: int

purpose: removes and returns the smallest valued element from the linked list stack \*/

node \*p = list;

int value = p->val;

list = p->next;

return value;

}

bool isEmpty()

{

/\* name: isEmpty

input: N/A

output: bool

purpose: determines if the linked list stack is empty \*/

return (list == NULL) ? true : false;

}

void display()

{

/\* name: display

input: N/A

output: N/

purpose: prints the contents of the list \*/

node \*p = list;

while (p != NULL)

{

std::cout << p->val << '\t';

p = p->next;

}

}

};

// function prototypes

OrderedList makeStack(int \*arr, int size);

OrderedList unionStacks(OrderedList setA, OrderedList setB);

int main()

{

/\* name: main

input: N/A

output: N/A

purpose: main function to drive the program \*/

// given sets a and b

int a[] = {3, 8, 4, 1};

int b[] = {4, 8, 6, 5, 7};

OrderedList setA = makeStack(a, int(sizeof(a)/sizeof(int)));

OrderedList setB = makeStack(b, int(sizeof(b)/sizeof(int)));

std::cout << "Set A=\t\t";

setA.display();

std::cout << std::endl;

std::cout << "Set B=\t\t";

setB.display();

std::cout << std::endl;

OrderedList setUnion = unionStacks(setA, setB);

std::cout << "A union B =\t";

setUnion.display();

std::cout << std::endl;

system("pause");

return 0;

}

OrderedList makeStack(int \*arr, int size)

{

/\* name: makeStack

input: int array a

output: OrderedList

purpose: creates an ordered stack list from an int array. \*/

OrderedList stack = OrderedList();

for (int i = 0; i < size; ++i)

{

stack.push(arr[i]);

}

return stack;

}

OrderedList unionStacks(OrderedList setA, OrderedList setB)

{

/\* name: unionStacks

input: OrderedList setA, OrderedList setB

output: OrderedList

purpose: creates an ordered stack list from the union of two other sets. \*/

OrderedList union\_ = OrderedList();

int valA = setA.pop();

int valB = setB.pop();

while (!setA.isEmpty() && !setB.isEmpty())

{

// traverses both lists to determine which values should enter the union.

// we can't insert every value from both lists because the union should not have duplicates.

// we compare the smallest values (valA and valB) of each set (setA and setB) and insert the

// smallest of those two values. Then we look at the next smallest value from its set and

// continue comparing and inserting. In the event that two values are equal, we insert only

// one of them and move to the next value.

// ex: setA = 1, 3, 4, 8 setB = 4, 5, 6, 7, 8, 9

// compare 1 and 4 -> insert 1, increment setA

// compare 3 and 4 -> insert 3, increment setA

// compare 4 and 4 -> insert 4, increment setA and setB

// compare 8 and 5 -> insert 5, increment setB

// compare 8 and 6 -> insert 6, increment setB

// compare 8 and 7 -> insert 7, increment setB

// compare 8 and 8 -> insert 8, increment setA and setB

// end while loop.

if (valA == valB)

{

union\_.push(valA);

valA = setA.pop();

valB = setB.pop();

}

else if (valA < valB)

{

union\_.push(valA);

valA = setA.pop();

}

else

{

union\_.push(valB);

valB = setB.pop();

}

}

// the above while loop does not compare the last values of both sets, so we must

// compare them manually

if (valA != valB) { union\_.push(valA); }

union\_.push(valB);

// if one of the sets empties before the other (ie. one set is larger than the other).

// then add the rest of the values of the larger set into the union.

// in the above example, notice that setB's last value, 9, is still not added into the union

// because setA emptied first and cause the while loop to end.

// in this event, we'll add the rest of the elements from setA into the union. but because it's

// empty, nothing is added. then we add the rest of the elements from setB into the union which

// will insert setB's last value, 9.

while (!setA.isEmpty())

{

valA = setA.pop();

if (valA != valB) { union\_.push(valA); }

}

while (!setB.isEmpty())

{

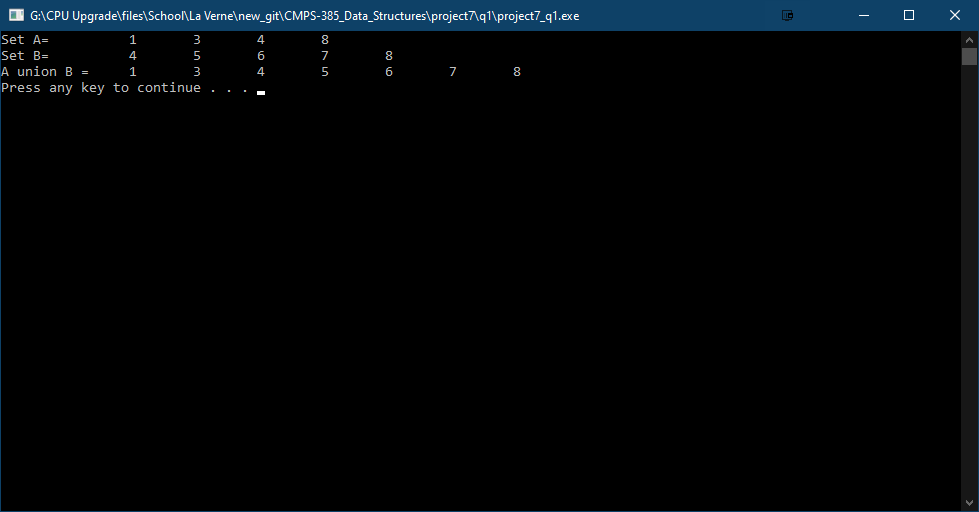
valB = setB.pop();

if (valA != valB) { union\_.push(valB); }

}

return union\_;

}



/\* -----------------------------------------------

Name: J-Zach Loke

Course: CMPS-385

Semester: Spring 2020

Project: No. 7 Part 2

Purpose: From a text file of names sort them using linked lists.

----------------------------------------------- \*/

#include <iostream>

#include <string>

#include <fstream>

class OrderedList

{

private:

struct node

{

std::string val;

node \*next;

};

node \*list;

public:

OrderedList() { list = NULL; }

void push(std::string x)

{

/\* name: push

input: std::string x

output: N/A

purpose: adds an element into the linked list stack and makes sure to keep it ordered. \*/

// create the node element we wish to insert

node \*insert = new(node);

insert->val = x;

// temporary variables to help traversal

node \*prev = list; // represents the node we've previously examined (needed because we can't traverse backwards).

node \*curr = list; // represents the current node we're examining.

while (true)

{

// determines where in the list to insert the new node.

if (curr == NULL)

{

// executes if we've reached the end of the list.

// if this executes, that means all nodes in the list have a smaller value than the node we're inserting.

// in this case, we should append the new node to the end of the list.

// also executes if the list is empty (ie. the first element is NULL)

if (list == NULL) { list = insert; }

else { prev->next = insert; }

insert->next = NULL;

return;

}

else if (curr->val.compare(insert->val) > 0)

{

// executes if the node we're examining has a value larger than the node we're inserting.

// if this executes, that means we've found the correct place to insert the new node.

if (prev == curr) { list = insert; }

else { prev->next = insert; }

insert->next = curr;

return;

}

// executes when we haven't found the correct place to insert the new node.

prev = curr;

curr = curr->next;

}

}

std::string pop()

{

/\* name: pop

input: N/A

output: std::string

purpose: removes and returns the smallest valued element from the linked list stack \*/

node \*p = list;

std::string value = p->val;

list = p->next;

return value;

}

bool isEmpty()

{

/\* name: isEmpty

input: N/A

output: bool

purpose: determines if the linked list stack is empty \*/

return (list == NULL) ? true : false;

}

void display()

{

/\* name: display

input: N/A

output: N/

purpose: prints the contents of the list \*/

node \*p = list;

while (p != NULL)

{

std::cout << p->val << std::endl;

p = p->next;

}

}

};

int main()

{

/\* name: main

input: N/A

output: N/A

purpose: main function to drive the program \*/

OrderedList presidents = OrderedList();

// given input file

std::fstream file;

file.open("input.txt", std::ios::in);

while (!file.eof())

{

std::string name;

file >> name;

presidents.push(name);

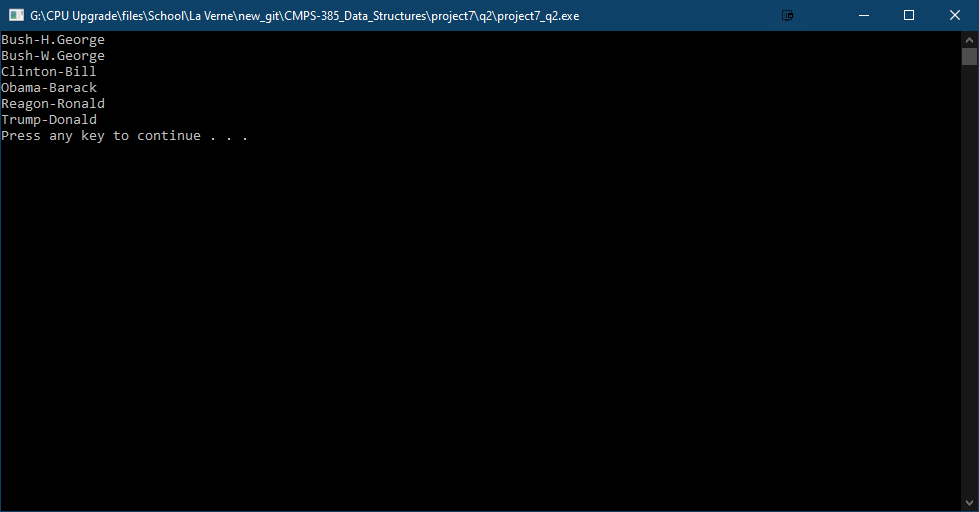
}

presidents.display();

system("pause");

return 0;

}



Question 3

1. Insert 10 in front of the list.

node \*insert = new(node);

insert->info = 10;

insert->next = list;

list = insert;

1. Delete the last node.

node \*curr = list;

node \*prev = list;

while (curr->next != NULL)

{

prev = curr;

curr = curr->next;

}

delete curr;

prev->next = NULL;

1. Insert 50 before the node whose info is 70.

node \*insert = new(node);

insert->info = 50;

node \*curr = list;

node \*prev = list;

while (curr->info != 70)

{

prev = curr;

curr = curr->next;

}

insert->next = curr;

prev->next = insert;

1. Insert 100 at the rear of the list.

node \*insert = new(node);

insert->info = 100;

insert->next = NULL;

node\* curr = list;

while (curr->next != NULL) { curr = curr->next; }

curr->next = insert;

1. Delete the node whose info is 60.

node \*curr = list;

node \*prev = list;

while (curr->info != 60)

{

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

delete curr;

1. Determine the number of nodes in the linked list.

int count = 0;

node \*index = list;

while (index != NULL)

{

index = index->next;

count++;

}

return count;